A Model for Managing Collections of Patterns

Baptiste Jeudy, Christine Largeron, and François Jacquenet

Laboratoire Hubert Curien, UMR CNRS 5516
Univ. of St-Etienne, France

Abstract. Data mining algorithms are now able to efficiently deal with huge amount of data. Various kinds of patterns may be discovered and may have some great impact on the general development of knowledge. In many domains, end users may want to have their data mined by data mining tools in order to extract patterns that could impact their business. Nevertheless, those users are often overwhelmed by the large quantity of patterns extracted in such a situation. Moreover, some privacy issues, or some commercial one may lead the users not to be able to mine the data by themselves. Thus, the users may not have the possibility to perform many experiments integrating various constraints in order to focus on specific patterns they would like to extract. Post processing of patterns may be an answer to that drawback. Thus, in this paper we present a framework that could allow end users to manage collections of patterns. We propose to use an efficient data structure on which some algebraic operators may be used in order to retrieve or access patterns in pattern bases.

1 Introduction

The amount of information that has been stored in data bases all around the world has continously increased among the years. In order to explore these potential mines of knowledge, efficient data mining tools have been designed for many years. Hence, it is now possible to mine huge databases in order to extract various kinds of patterns, modeling some knowledge. Depending on the algorithms used by end users for their needs, patterns may be varied, we may cite for example decision trees, association rules, formal concepts, etc. While mining huge databases is becoming a common task for many users, those one are now faced with a new problem: how can they exploit the large amount of patterns that are commonly extracted by the data mining tools. Indeed, in the same way it was impossible to manually extract knowledge from huge databases, it is now impossible to manage large volumes of patterns and the end users are in need of new tools in order to do that.

In fact two approaches have been proposed to users in order to manage and explore what is commonly called Pattern Bases. The first one is based on the concept of inductive databases [8,2,13,12]. In Europe, the CInQ project[1] has played a dynamic role in researches in that domain. An inductive database not

[1] http://www.cinq-project.org/

only contains data but also patterns and data mining languages integrated in the inductive database management systems offer some facilities for pattern manipulation through post-processing operators [3]. Nevertheless those one are very basic and pattern base management systems should provide more sophisticated functionalities.

The second approach for managing patterns focuses on Pattern Base Management Systems (PBMS). In [5,4], a PBMS is defined as "a system for handling (storing / processing / retrieving) patterns defined over raw data in order to efficiently support pattern matching and to exploit pattern-related operations generating intentional information". Thus, the principle consists in storing the patterns extracted by some data mining systems using some efficient data structures. Pattern manipulation languages have then to be designed in order to manage them. This approach involves two questions. The first one concerns the possibility to design a generic model for patterns, the second one concerns the language needed to access and query patterns. The PANDA project[2] [10] is an interesting work in that way. It proposes a generic framework to model various classes of patterns, then some SQL-like operators allow the user to manage them. Nevertheless, as the underlying model used for storing the patterns is the relational model, the requests that can be designed by users are very complex, non intuitive and time consuming. Even if SQL may be considered an obvious candidate to manage collections of patterns, it was in fact designed to access data stored in databases and it is not well suited to manage patterns [11]. Zaki also proposed in [18] a generic framework for specifying data structures and management functionalities on patterns. Tuzhilin [15] specifies some SQL-like operators in order to explore sets of association rules. In those two cases, while some efforts have been done in order to efficiently store patterns, the languages proposed to handle them are quite poor. Finally, in the field of pattern base management, we may cite the PMML project [7] that allows interoperability of pattern bases, specifying an XML framework associated to the concept of pattern. Nevertheless this framework is more concerned with structured representation of patterns than with their management.

Our work also belongs to this second approach based on the post processing of patterns. That is we aim at designing a data structure and efficient algorithms for the management of large pattern bases. We think that it may be interesting for the users to be able to get various sets of patterns, that could be successively extracted running data mining tools on various databases, and then to use efficient tools to manage them. Indeed, in many cases, due to privacy issues or commercial one, the user does not have any access to the data. In this paper, we propose a framework for the management of a particular class of patterns that are called concepts [16]. More precisely, our approach is based on labeled graphs to represent collections of concepts. In this domain few works have been done. The most related one to ours is probably the work of Mielikäinen [9] who

[2] http://dke.cti.gr/panda

suggested to represent patterns using deterministic finite automata. The results obtained experimentally show that minimum automata provide a compact representation. Nevertheless, Mielikäinen considered collections of itemsets and not of concepts. Moreover he does not provide any generic framework, based on some algebraic operators.

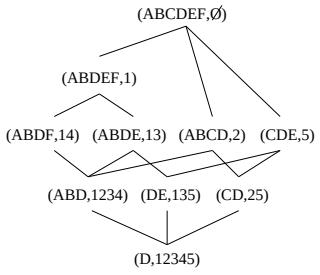
The next section recalls some basic definitions useful for the understanding of the paper. In Section 3, we introduce the labeled graph representation of concepts collections while Section 4 presents a basic algorithm to build this graph. In Section 5 we define operators that allow to query the graph and that can be combined using an algebra (in some sense, this section is related to [6]).

2 Definitions

A database Db is a relation between a **set of attributes** $\mathcal{A} = \{a_1, a_2, ...\}$ and a **set of objects** $\mathcal{O} = \{o_1, o_2, ...\}$.

Such a database can be represented as a boolean matrix where the columns are attributes and the rows are objects.

	A	B	C	D	E	F
1	1	1	0	1	1	1
2	1	1	1	1	0	0
3	1	1	0	1	1	0
4	1	1	0	1	0	1
5	0	0	1	1	1	0



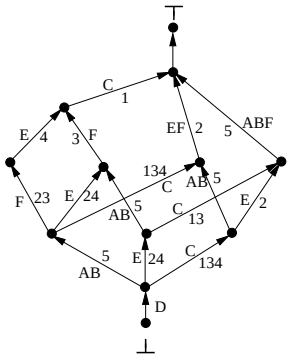


Fig. 1. Example of a database where $\mathcal{A} = \{A, B, C, D, E, F\}$ and $\mathcal{O} = \{1, 2, 3, 4, 5\}$ (top), Hasse diagram of the formal concept collection $\mathsf{Concepts}(Db)$ (left) and the corresponding graph representation with labels on the edges (right, see Sect. 3)

For instance, this database can be the result of gene expression measures. In this case, the columns represent genes and the rows represent biological situations. There is a relation between a gene and a situation if the gene is overexpressed in the given situation. Mining formal concepts in this kind of data has been shown to be interesting for biologists [1].

A **bi-set** is a pair (X, Y) where $X \subseteq \mathcal{A}$ and $Y \subseteq \mathcal{O}$. A **1-rectangle** is a bi-set (X, Y) such that all the attributes of X are in relation with all the objects of Y. In the matrix, a 1-rectangle thus defines a sub-matrix containing only ones.

Example 1. In our example of Fig. 1, $(ABD, 123)$ (we use this notation for $(\{A, B, D\}, \{1, 2, 3\})$) and $(E, 135)$ are 1-rectangles. $(ABC, 12)$ is a bi-set but is not a 1-rectangle since C is not in relation with 1.

The inclusion $\subseteq$ on bi-sets is defined by: $(X_1, Y_1) \subseteq (X_2, Y_2)$ iff $X_1 \subseteq X_2$ and $Y_1 \subseteq Y_2$. A **formal concept** is then a maximal 1-rectangle for the order defined on bi-sets by the inclusion. The collection of all formal concepts in a database Db is $\mathsf{Concepts}(Db)$ (see Fig. 1).

We then define an **order on the concepts** as follow: $(X, Y) \preceq (X', Y')$ iff $X \subseteq X'$ and $Y' \subseteq Y$ (notice the direction of the inclusion). With this order, the collection of formal concepts forms the well known formal concept lattice. The Hasse diagram of this lattice for our running example is presented in Fig. 1 (left).

3 Representation of a Collection of Concepts

There are several desirable properties for a good representation:

- The representation must allow querying: for instance, given a collection $\mathcal{C}$ of concepts, we want to be able to select all concepts containing a given attribute or object, or all the concepts containing at least 5 objects...
- The result of a query must be a collection of concepts with the same representation as the original collection (closure property). This is important to support successive queries on a collection.
- In the definitions, there is a duality between objects and attributes. The representation should respect this duality. If it is the case, we can use "dual" algorithms for dual operations. For instance, the algorithm to select all concepts containing a given attribute will be the dual of the algorithm selecting all concepts containing a given object.

The output of concept extraction algorithms (such as D-miner [1]) is typically a file containing a list of concepts. This is probably the most simple way to represent a collection of concepts.

Mielikäinen [9] proposed to use an automaton to store an itemset collection (an itemset is a set of attributes). Several automata are possible: for instance a simple prefix tree or a minimum automaton. However, it is necessary to define an order on the attributes to transform itemsets into strings and choosing a good ordering is very difficult [9] and not very natural. Using an automaton to represent concepts is also possible if we can transform concepts into strings. However, doing this without introducing a arbitrary order or losing the duality between objects and attributes seems very difficult.

To solve the problem of the need to choose an order, Mielikainen proposed to use what he called commutative automata [9]. However, these automata have a

lot of edges and this is an issue if we want to query efficiently the representation. Furthermore, the commutative automata only store the attributes of concepts (and not the objects). This means that the duality is of course lost and that it will be impossible to query the set of objects of the concepts without recomputing them.

We propose to use a labeled graph: the Hasse diagram of the order $\preceq$ on the collection of concepts: the vertices are the concepts and there is an edge $X \to Y$ between the concepts X and Y iff Y cover X, i.e., $X \prec Y$ and it does not exist a concept Z such that $X \prec Z \prec Y$. We add two special vertices: $\bot$ and $\top$ such that $(X, Y) \prec \top$ and $\bot \prec (X, Y)$ for all (X, Y).

We can choose to put the labels on the edges or on vertices: On the vertices: the label consists of the two sets X and Y. On the edges: on the edge $(X, Y) \to (X', Y')$ the label consists of the sets $X' \setminus X$ and $Y \setminus Y'$.

Figure 1 shows an example of the constructed graph with the collection of all the concepts in the database.

With this representation, we do not need to order the attributes or the objects and we will show that it is easy to query this representation.

4 Construction of the Graph Representation

Given a list of concepts extracted by a concept extraction algorithm such as D-miner [1], the following algorithm constructs the graph representing the collection. In fact this algorithm is a common release of classical algorithms that have been investigated by the Formal Concept Analysis community [17] in order to build a graph representation of concepts. As this is not the core of our paper, we do not provide too much details on this construction.

The idea of the construction of the graph is to start from a graph representing the empty collection (which contains only the vertices $\top$ and $\bot$) and to insert the other concepts in the graph one after the other. In order to simplify the algorithm, we choose to add the concepts (X, Y) in order of the increasing size of X.

When a new concept $C = (X, Y)$ is inserted, there is no other concept C' in the graph such that $C \preceq C'$ (because of the order in which the concepts are inserted). Therefore, the only successor of C is $\top$ and an arc $C \to \top$ is added. Next, we must find all predecessors C' of C in the graph (i.e., the concepts C' in the graph such that C covers C') to create the arcs $C' \to C$.

For this purpose, a depth first traversal of the graph is performed (starting from $\top$). The whole graph does not need to be traversed: each time that a concept C' covered by C is found, there is no need to explore the concepts smaller than C' (for $\preceq$) since none of them can be covered by C.

Finally, if C covers a concept C' that was covered by $\top$, the edge $C' \to \top$ must be removed (since $\top$ no longer covers C').

This is implemented by the algorithm `construct_graph`. It uses functions to manipulate the graph (`insert_vertex`, `insert_edge` and `delete_edge` which are not detailed) and call a procedure `insert_concept` to insert the next concept

in the graph. This procedure call a recursive procedure `rec_insert` to traverse the graph (the set E is used to "mark" the vertices that have been explored).

Algorithm 1: construct_graph

```
Input: An ordered collection C of concepts
Output: A graph G representing the collection C
G = empty_graph
insert_vertex(⊤, G)
insert_vertex(⊥, G)
insert_edge(⊥ → ⊤, G)
forall B ∈ C do
    insert_concept(B, G)
return G
```

Procedure `insert_concept`(*concept B, graph G*)

```
insert_vertex(B, G)
E = ∅                                          // E is a global variable
forall X ∈ predecessor(⊤) do
    if X ⪯ B then
        delete_edge(X → ⊤, G)
        insert_edge(X → B, G)
    else
        rec_insert(B, X, G)
insert_edge(B → ⊤, G)
```

Procedure `rec_insert`(*concept B, vertex V, graph G*)

```
forall X ∈ predecessor(V) \ E do
    E = E ∪ {X}
    if X ⪯ B then
        if ∄Y ∈ successor(X) such that Y ⪯ B then
            insert_edge(X → B, G)
    else
        rec_insert(B, X, G)
```

5 Queries

In this section, we study different operations that can be made on a collection of concepts. We distinguish two kind of queries: selection and projection queries.

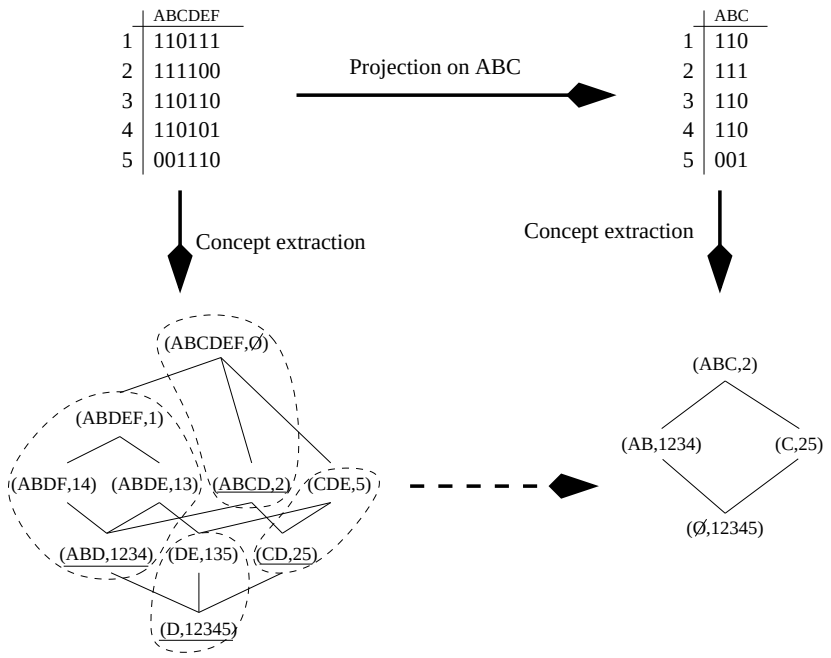


Fig. 2. Projection of concepts. The original database Db and the corresponding concepts $\mathsf{Concepts}(Db)$ (left); The projected database $\pi_{\{A,B,C\}}(Db)$ and $concept(\pi_{\{A,B,C\}}(Db))$ (right). The $\{A, B, C\}$-equivalence classes (dotted, see Def. 1) and their least elements (underlined). One can check the fact that the intersection of the least elements with $\{A, B, C\}$ are exactly the concepts of $\pi_{\{A,B,C\}}(Db)$ (Theorem 1).

5.1 Selection Queries

Given a collection of concepts $\mathcal{C}$ and a predicate p on the concepts, we define the selection with respect to p as

$$\sigma_p(C) = \{(X, Y) \in C \mid p(X, Y) \text{ is true}\}$$

Example 2. Classical examples of selection predicates include [14]:

- minimum (or maximum) length: $p(X, Y) = (|X| > \gamma)$
- minimum (or maximum) frequency: $p(X, Y) = (|Y| > \gamma)$
- minimum (or maximum) area: $p(X, Y) = (|X|.|Y| > \gamma)$.
- requiring that an attribute (object) belongs (does not belong) to a concept: $p(X, Y) = (A \in X)$.
- ...

5.2 Projection Queries

For example, given gene expression data, a biologist might be interested in only a part of the genes. He may want to focus only on a subset of the genes, for instance the genes A,B and C.

The most simple solution would be to extract the concepts not on the whole dataset, but on a part of it containing only the columns A, B and C, i.e., on a projection of the original database (see Fig. 2, right). If A is a set of attributes, we denote $\pi_A(Db)$ the projection of the database Db on the attributes of A.

However, a new extraction of concepts in the projected database would be expensive. Furthermore, the original data are perhaps not available anymore (for privacy purposes for example). If the collection of concepts in the whole database is still available, a natural question is whether it is possible to compute the collection of concepts in the projected database from the concepts in the whole database (i.e., to find the operation corresponding to the dotted arrow in Fig. 2).

In other words, we want to be able to compute $\mathsf{Concepts}(\pi_A(Db))$ from $\mathsf{Concepts}(Db)$ without having to perform an extraction in $\pi_A(Db)$. It is indeed possible. First, we need to define an A-equivalence relation on the concepts.

Definition 1 (A-equivalence). *Given a set A of attributes, two concepts (X, Y) and (X', Y') are A-equivalent iff $X \cap A = X' \cap A$.*

This is obviously an equivalence relation. Figure 2 gives an example of the equivalence classes. Furthermore, we have the following proposition:

Proposition 1. *The A-equivalence classes have a least element (for $\preceq$).*

To prove this proposition, we use the following well known result: if $C_1 = (X_1, Y_1)$ and $C_2 = (X_2, Y_2)$ are two concepts, then there exists a concept $C = (X_1 \cap X_2, Y)$ with $Y_1 \cup Y_2 \subseteq Y$.

Proof. Given two A-equivalent concepts $C_1 = (X_1, Y_1)$ and $C_2 = (X_2, Y_2)$, then there exists a third concept $C = (X_1 \cap X_2, Y)$ with $Y_1 \cup Y_2 \subseteq Y$.

Of course, C is A-equivalent to C_1 and C_2 and we also have $C \preceq C_1$ and $C \preceq C_2$ (by definition of $\preceq$).

Therefore, the A-equivalence class of C_1 and C_2 has only one minimum element, i.e., it has a least element.

□

The following theorem characterizes the collection $\mathsf{Concepts}(\pi_A(Db))$ with respect to $\mathsf{Concepts}(Db)$.

Theorem 1. *Given a database Db and a set of attributes A, we denote by LE_A the set of the least elements of the A-equivalence classes. Then*

$$\mathsf{Concepts}(\pi_A(Db)) = \{(X \cap A, Y) \mid (X, Y) \in \mathrm{LE}_A\}.$$

Proof. In this proof, we use the fact that if (X, Y) is a concept in $\pi_A(Db)$ then it can be "extended" to form a concept (X', Y) in Db where $X' \cap A = X$.
First inclusion $\subseteq$:
Let (X, Y) be a concept in $\pi_A(Db)$. We can "extend" it to a concept (X', Y) of Db. Let (X'', Y'') be a concept A-equivalent to (X', Y) such that $(X'', Y'') \preceq (X', Y)$.

$(X'' \cap A, Y'') = (X, Y'')$ is a 1-rectangle of $\pi_A(Db)$. Since $Y \subseteq Y''$ and (X, Y) is a concept of $\pi_A(Db)$, Y and Y'' are equal. Therefore (X'', Y'') is included in (X', Y) and therefore $X'' = X'$ which means that $(X'', Y'') = (X', Y)$ and (X', Y) is the least element of its A-equivalence class.
Inclusion $\supseteq$:
Let $(X, Y) \in \mathrm{LE}_A$. Then $(X \cap A, Y)$ is a 1-rectangle in $\pi_A(Db)$. Suppose that there exists a 1-rectangle (X', Y') in $\pi_A(Db)$ such that $(X', Y') \supseteq (X \cap A, Y)$. Then $X' = X \cap A$ otherwise $(X \cup X', Y)$ is a 1-rectangle strictly containing (X, Y) and therefore (X, Y) cannot be a concept. We can extend $(X', Y') = (X \cap A, Y')$ to a concept (X'', Y') of Db. Then $X'' \subseteq X$ otherwise $(X \cup X'', Y)$ is a 1-rectangle strictly containing (X, Y) and thus (X, Y) cannot be a concept. Therefore $(X'', Y') \preceq (X, Y)$ and these two concepts are A-equivalent. Therefore they are equal (since (X, Y) is a least element) and $(X \cap A, Y)$ is a maximal 1-rectangle in $\pi_A(Db)$ (for $\subseteq$), i.e., a concept of $\pi_A(Db)$. □

More generally, we can define a projection operation on collections of concepts:

Definition 2 (collection of concepts projection).
Given a collection of concepts $\mathcal{C}$ in a database Db and a set of attributes A, we define the projection of the collection $\mathcal{C}$ with respect to A by:

$$\pi_A(\mathcal{C}) = \{(X \cap A, Y) \mid (X, Y) \in \mathcal{C} \cap \mathrm{LE}_A\}$$

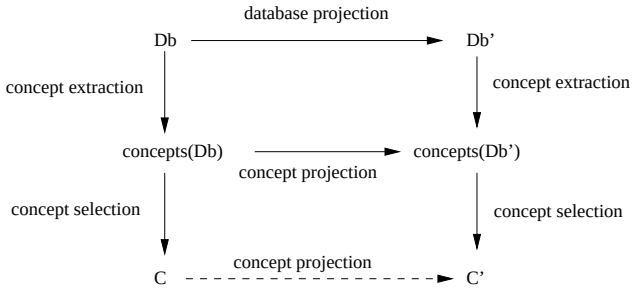
where LE_A is defined as in Theorem 1.

Theorem 1 means that this projection operation can be used to compute the concepts in the projected database $\pi_A(Db)$ by projecting the concepts of the original database Db: $\mathsf{Concepts}(\pi_A(Db)) = \pi_A(\mathsf{Concepts}(Db))$. In this equality, the first π_A denotes a database projection whereas the second one denotes a collection of concepts projection (Def. 2).

5.3 Algebra

In this section, we study how the projection and selection operations on collection of concepts compose with each other.

We want to know if there exists an operation to close the following diagram (dotted arrow). A natural candidate is the projection that we have just defined.



Indeed, the following theorem shows that this diagram can be closed using the projection operation:

Theorem 2. *Given a collection of concepts $\mathcal{C}$ in a database Db, a set of attributes A and a selection predicate p such that for all concepts (X, Y), $p(X \cap A, Y) = p(X, Y)$, then*

$$\pi_A \circ \sigma_p(\mathcal{C}) = \sigma_p \circ \pi_A(\mathcal{C}).$$

Proof. $(X, Y) \in \pi_A(\sigma_p(\mathcal{C})) \iff \exists (X', Y) \in \sigma_p(\mathcal{C}) \cap \mathrm{LE}_A$ such that $X = X' \cap A$ (by Def. 2) $\iff \exists (X', Y) \in \mathrm{LE}_A \cap \mathcal{C}$ such that $p(X', Y)$ is true and $X = X' \cap A \iff \exists (X', Y) \in \mathrm{LE}_A \cap \mathcal{C}$ such that $p(X, Y)$ is true and $X = X' \cap A$ (since $p(X', Y) = p(X' \cap A, Y) = p(X, Y)$) $\iff (X, Y) \in \pi_A(\mathcal{C})$ and $p(X, Y)$ is true (by Def. 2) $\iff (X, Y) \in \sigma_p(\pi_A(\mathcal{C}))$ □

The requirement on p can seem very strong but it is necessary. In order to be able to perform the selection after the projection, the projection must not remove too much information from the collection. For instance, if the selection is defined by $p(X, Y) = (D \in X)$ (i.e., select the concepts containing attribute D), then this selection does not commute with the projection $\pi_{\{A,B,C\}}$. Indeed, after this projection the information whether a concept contained attribute D is no longer available. There is a similar behavior with selection and projection defined on relational tables. If the selection uses an attribute which is suppressed by the projection, the two operations do not commute.

5.4 Duality

In the two previous sections, we defined the projection of a collection of concepts on a set A of attributes. In a dual manner, we can define another projection on a set O of objects. The dual equivalence relation of the A-equivalence (Def. 1) can be defined as follow: two concepts (X, Y) and (X', Y') are O-equivalent iff $Y \cap O = Y' \cap O$. Then we have the dual of theorems 1 and 2.

5.5 Algorithms

In this section, we present the algorithms to actually perform the selection and projection on the graph representation of the collection.

To perform the projection of a collection of concepts $\mathcal{C}$ with respect to a set of attributes A, we need to be able to test if a concept is minimal in its A-equivalence class. However, this is not always possible without additional information: it is possible that the collection $\mathcal{C}$ does not contain all the concepts belonging to an equivalence class, in this case, we could find a minimum concept in this equivalence class in $\mathcal{C}$ which is not the least element of this equivalence class in $\mathsf{Concepts}(Db)$.

For instance, suppose $\mathcal{C}$ contains all the concepts of Fig. 2 (before projection) except concept $(D, 12345)$. Then, if we compute the projection of this collection with respect to $\{A, B, C\}$, we must be able to detect that $(DE, 135)$ is not a least element of an equivalence class. Without additional information, it is not possible without a possibly expensive check in the data.

This is the reason why we add some information in our graph representation. Given a collection $\mathcal{C}$ of concepts, we add into the graph the concepts that are "just outside" of the collection. By just outside, we mean the concepts that are either predecessor or successor of a concept belonging to the collection. These additional concepts are marked and are not linked to $\top$ and $\bot$ (they are inserted in the graph with the

`insert_marked_vertex` function), they are linked only to the concept(s) of the collection which is (are) their predecessor or successor. Of course, when doing selection or projection operations, this additional information must be maintained.

The algorithm to perform the projection is given in Alg. 6. For all vertex X, the algorithm computes `le[X]` which is the least element of the A-equivalence class of X. If this least element is not in the collection, then `le[X]=NIL`. The least elements of the equivalence classes are inserted in the new graph G' and the edges are added to G'. In the algorithm, we use the notation `proj(X,Y)` to denote $(X \cap A, Y)$.

Algorithm 4: selection_AM

Input: A graph G representing a collection $\mathcal{C}$ of concepts and an anti-monotonic selection predicate p
Output: A graph G' representing the collection $\sigma_p(\mathcal{C})$
G' = empty_graph
`insert_vertex`($\top$, G')
`insert_vertex`($\bot$, G')
$E = \emptyset$ `// E is a global variable`
explore($\bot$)
return G'

In the general case, to compute the selection of a collection of concepts with respect to a predicate p, we must traverse the graph representing the collection and test p on all concepts.

However, when p is monotonic or anti-monotonic, it is not necessary to traverse the whole graph. A predicate p is anti-monotonic iff $(\neg p(X,Y) \wedge ((X,Y) \preceq (X',Y'))) \Rightarrow \neg p(X',Y')$ and monotonic iff $(\neg p(X,Y) \wedge ((X',Y') \preceq (X,Y))) \Rightarrow \neg p(X',Y')$. Therefore, if p is anti-monotonic, the graph can be explored bottom up (from $\bot$ to $\top$) and if a concept X that does not satisfy p is found, it is not necessary to explore its successors (see Alg. 4). Dually, for a monotonic constraint, the graph is explored top down.

Procedure `explore(`*vertex* V`)`

```
E = E ∪ {V}                                          // E is a global variable
forall X ∈ predecessor(V) and X marked do
    insert_marked_vertex(X, G')
    insert_edge(X → V, G')
link_to_top = true
forall X ∈ successor(V) do
    if p(X) and X not marked then
        link_to_top = false
        if X ∉ E then
            insert_vertex(X, G')
            explore(X)
        insert_edge(V → X, G')
    else
        insert_marked_vertex(X,G')
        insert_edge(V → X, G')
if link_to_top then
    insert_edge(V → ⊤)
```

Algorithm 6: projection

```
Input: A graph G representing a collection C of concepts and a set A of
       attributes
Output: A graph G' representing the collection π_A(C)
forall X ∈ C, X not marked do
    le[X]= X
forall X ∈ C, X not marked, in topological order do
    if le[X] = X then                                   // X is perhaps in LE_A
        if ∃Y ∈ predecessor(X), Y marked and class(Y) = class(X) then
        // X is not in LE_A
            le[X] = NIL
        else                                                  // X is in LE_A
            insert_vertex(proj(X), G')
            forall X' marked ∈ predecessor(X) do
                insert_marked_vertex(proj(X'), G')
                insert_edge(proj(X') → proj(X), G')
    forall Y unmarked ∈ successor(X) do
        if class(Y) = class(X) then
            le[Y] = le[X]
forall edge X → Y in G, X and Y unmarked do
    insert_edge(proj(le[X]) → proj(le[Y]), G')
```

6 Conclusion

In this article, we made an original study on how to represent and query collections of concepts. We proposed to store these collections using a graph representation and we defined two kinds of operators: selection and projection.

We want to extend this work in several directions. First, it would be interesting to study the scalability of our representation on real datasets and make comparison with, for instance, automata representations. Studying the relationships between the size of the representation and the characteristics of the datasets from which it was extracted would also be interesting.

Second, our representation using a graph is efficient for querying but is could be more compact. One could use two representations of the collection of concepts: a very compact one for long term storage (on disk) and another one (the graph) for querying.

Finally, several works have been done on generalization of concepts and on clustering of concepts. It would be interesting to study if it is possible to define an aggregation operator (a kind of "group by" operator) on the graph to support these generalization facilities..

7 Acknowledgments

This work is partially funded by the french ACI "masse de données" (Bingo project).

References

1. J. Besson, C. Robardet, J.-F. Boulicaut, and S. Rome. Constraint-based concept mining and its application to microarray data analysis. *IDA*, 9(1):59–82, 2005.
2. J.-F. Boulicaut, M. Klemettinen, and H. Mannila. Modeling KDD processes within the inductive database framework. In *DaWaK*, volume 1676 of *LNCS*, pages 293–302, 1999.
3. J. F. Boulicaut and C. Masson. Data mining query languages. In *The Data Mining and Knowledge Discovery Handbook*, pages 715–727. Springer, 2005.
4. B. Catania and A. Maddalena. *Pattern Management: Practice and Challenges*, pages 280–317. Processing and Managing Complex Data for Decision Support. Idea Group Publishing, 2006.
5. B. Catania, A. Maddalena, M. Mazza, E. Bertino, and S. Rizzi. A framework for data mining pattern management. In *PKDD*, volume 3202 of *LNCS*, pages 87–98, 2004.
6. C. T. Diop, A. Giacometti, D. Laurent, and N. Spyratos. Computation of mining queries: An algebraic approach. In *Constraint-Based Mining and Inductive Databases*, volume 3848 of *LNCS*, pages 102–126, 2005.
7. R. L. Grossman, S. Bailey, A. Ramu, B. Malhi, P. Hallstrom, I. Pulleyn, and X. Qin. The management and mining of multiple predictive models using the predictive model markup language (PMML). In *Information and Software Technology*, volume 41, pages 589–595, 1999.
8. T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Comm. ACM*, 39(11):58–64, 1996.
9. T. Mielikäinen. An automata approach to pattern collections. In *KDID*, volume 3377 of *LNCS*, pages 130–149, 2004.
10. Panda. Patterns for next-generation database systems (2001-2004). FET/IST-2001-33058.
11. K. Parsaye. From datamagement to pattern management. *DM Rev. Mag.*, 1999.
12. L. D. Raedt. A perspective on inductive databases. *SIGKDD Explorations*, 4(2):69–77, 2002.
13. L. D. Raedt, M. Jaeger, S. Lee, and H. Mannila. A theory of inductive query answering. In *ICDM*, pages 123–130, 2002.
14. A. Soulet and B. Crémilleux. An efficient framework for mining flexible constraints. In *PAKDD*, volume 3518 of *LNCS*, pages 661–671, 2005.
15. A. Tuzhilin and B. Liu. Querying multiple sets of discovered rules. In *SIGKDD*, pages 52–60. ACM, 2002.
16. R. Wille. Concept lattices and conceptual knowledge systems. *Comp. math. applied*, 23(6-9):493–515, 1992.
17. R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.
18. M. J. Zaki, N. Parimi, N. De, F. Gao, B. Phoophakdee, J. Urban, V. Chaoji, M. A. Hasan, and S. Salem. Towards generic pattern mining. In *Proc. of the Conference on Formal Concept Analysis*, volume 3403 of *LNCS*, pages 1–20, 2005.